

Tecnologias e Programação de Sistemas de Informação

Herança e Polimorfismo

Arquitetura de Dispositivos | David Jardim

Cofinanciado por:



Da aula anterior...

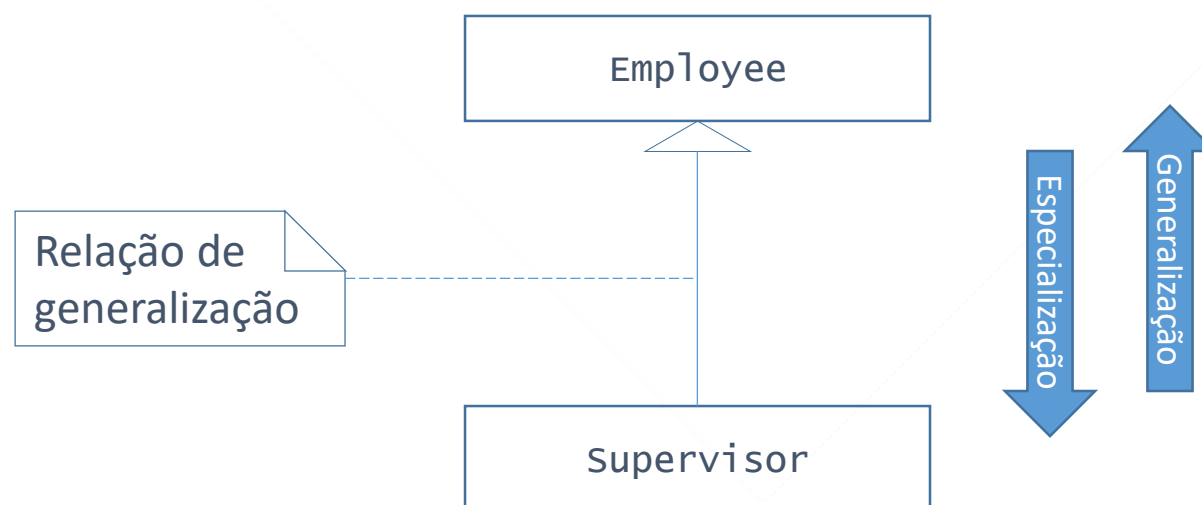
- Classes e Objetos
- Strings em Java

Employee

```
public class Employee {  
  
    private String name;  
    private String ssn;  
  
    public Employee(final String name, final String ssn) {  
        this.name = name;  
        this.ssn = ssn;  
    }  
    public String getName() {  
        return name;  
    }  
    public String getSsn() {  
        return ssn;  
    }  
    @Override  
    public String toString() {  
        return "(" + getName() + ", " + getSsn() + ")";  
    }  
}
```

Que é isto? Veremos à frente...

Generalização (relação)



- Um Supervisor é um Employee.
- Um Employee pode ser um Supervisor.

Herança

```
public class Supervisor extends Employee {  
  
    private int level;  
    public Supervisor(final String name, final String ssn, final int level) {  
  
    }  
  
    public int getLevel() {  
        return level;  
    }  
  
    @Override  
    public String toString() {  
        return "(" + getName() + ", " + getSsn() + ", "  
            + getLevel() + ")";  
    }  
}
```

Um Supervisor é um Employee.

Novo método específico da classe supervisor.

Sobrepõe-se ao método com a mesma assinatura na classe base Employee.

Generalização (relação)

Classe base ou superclasse.

Employee

Supervisor

Especialização

Generalização

Classe derivada ou subclasse.

Herança

- Classe derivada **deriva** da classe base (subclasse deriva da superclasse)
- Membros são herdados e mantêm categoria de acesso
- Relação **é um** – Referências do tipo da classe base podem referir-se a objectos de classes derivadas
- Exemplo

```
Supervisor supervisor = new Supervisor("Pedro", "123456789", 3);
```

```
Employee employee = new Supervisor("Ana", "987654321", 5);
```

Herança

- Classe derivada **tem todas as propriedades da base**
- Exemplo:

```
Supervisor supervisor = new Supervisor("Pedro", "12345", 3);
```

```
Employee employee = new Supervisor("Ana", "98765", 5);
```

```
String employee_ssn_id_1 = employee.getSsn();
```

```
String employee_ssn_id_2 = supervisor.getSsn();
```



Sobreposição

- Método de classe derivada pode **sobrepôr-se** a método de classe base
- Sobreposição é **especialização**
- Regras
 - Mesma assinatura e tipo de devolução compatível
 - Método na classe base não privado e não final
 - Método na classe derivada com acessibilidade igual ou superior

Na realidade tem de ser co-variante, ou seja, o tipo de devolução do método na classe derivada deriva de (ou é igual a) o tipo de devolução na classe base.

Um método final não pode ser especializado.

Categorias de acesso

- 
- Acessibilidade crescente
- Atributos ou membros podem ser
 - **private** – acesso apenas por outros membros da mesma **classe**
 - **package-private** (sem qualificador) – adicionalmente, acesso por membros de classes do mesmo **pacote**
 - **protected** – adicionalmente, acesso por membros de **classes derivadas**
 - **public** – acesso universal

Interfaces de uma classe

- Dentro da própria classe tem-se acesso a:
 - Membros da classe e membros não privados de classes base
- Nas classes do mesmo pacote tem-se acesso a:
 - Membros não privados da classe ou suas bases
- Numa classe derivada:
 - Membros protegidos ou públicos da classe ou suas bases
- Noutras classes:
 - Membros públicos da classe ou suas bases

Exemplo

```
Vector<Employee> employees =  
    new Vector<Employee>();
```

```
employees.add(new Employee("João Maria", "12345"));  
employees.add(new Supervisor("Ana Maria", "98765", 4));
```

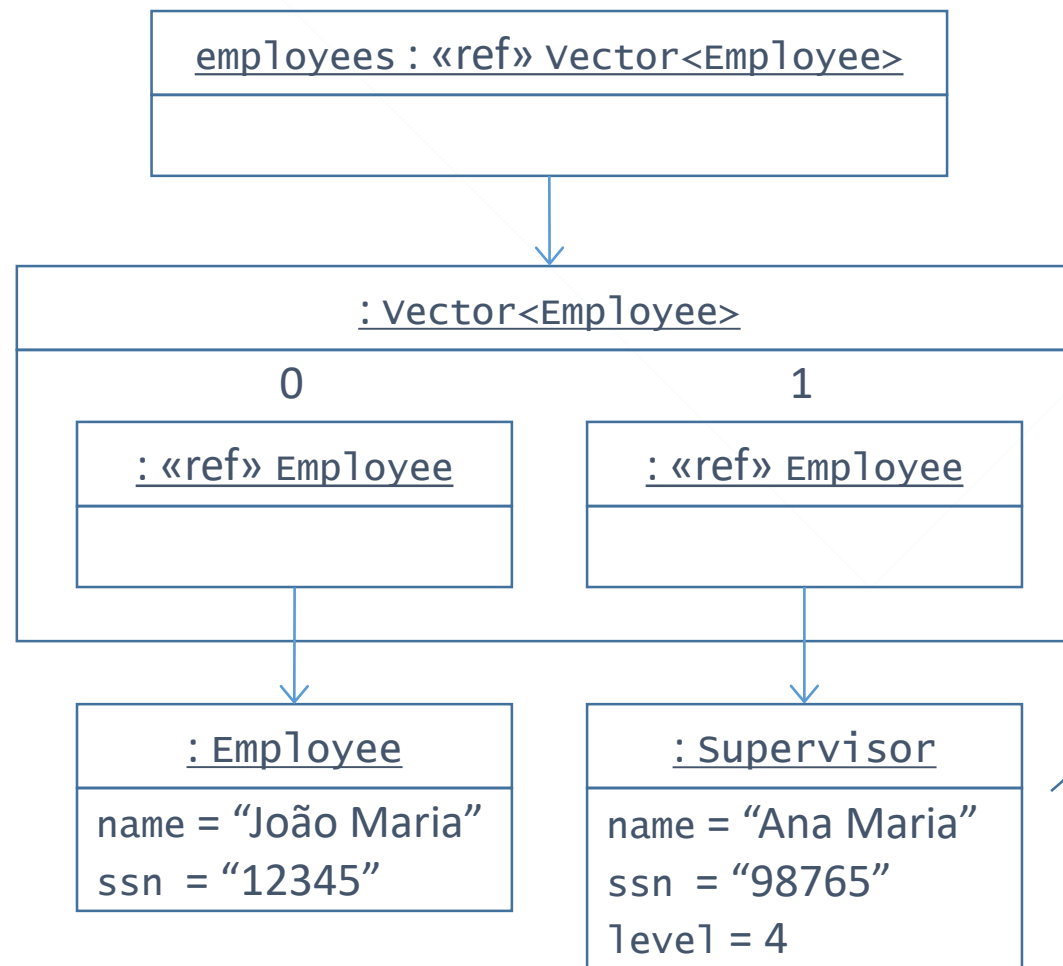
...

Qual o método toString() executado?

```
for (Employee employee : employees)  
    out.println(employee.toString());
```

Invocação da
operação
toString().

Organização



Possível porque a classe Supervisor deriva da classe Employee, ou seja, possível porque um Supervisor é (sempre também) um Employee.

Resultado

O resultado depende do tipo do objecto e não do tipo da referência! Isso acontece porque o método toString é polimórfico ou virtual.

(João Maria, 123456789)
(Ana Maria, 987654321, 4)

Polimorfismo

- Capacidade de um objecto tomar várias formas
 - A forma descrita pela classe a que pertence
 - As formas descritas pelas classes acima na **hierarquia** a que pertence
- Objecto pode ser referenciado por referências do tipo da classe a que pertence ou de classes acima na hierarquia (mais genéricas)

O que aparece na consola?

```
Supervisor supervisor = new Supervisor("Guilhermina", "12345", 3);  
Employee anEmployee = new Supervisor("Felisberto", "98765", 5);  
Employee anotherEmployee = new Employee("Elvira", "1111");  
out.println(supervisor.toString());  
out.println(anEmployee.toString());  
out.println(anotherEmployee.toString());
```

```
(Guilhermina, 123456789, 3)  
(Felisberto, 987654321, 5)  
(Elvira, 111111111)  
—
```


Polimorfismo: operações e métodos

- Uma **operação polimórfica** ou **virtual** pode ter várias implementações
- A uma implementação de uma operação chama-se método
- A uma operação polimórfica podem corresponder diferentes métodos, cada um em sua classe
- Todas as operações em Java são polimórficas, com excepção das qualificadas com `private`
- Uma classe é polimórfica se tiver pelo menos uma operação polimórfica

Polimorfismo: operações e métodos

- **Invoca**-se uma **operação** sobre um objecto de uma classe para atingir um objetivo
- Invocação de uma operação leva à **execução** do **método** apropriado, ou seja, leva à execução da **implementação** apropriada da operação
- Polimorfismo
 - Invocação de uma operação pode levar à execução de diferentes métodos
 - Método efectivamente executado depende da classe do objecto sobre o qual a operação é invocada
 - Método executado não depende do tipo da referência para o objecto utilizado

Simplificação... invocações internas podem levar à execução de métodos privados directamente.

A classe Object

```
public class Employee extends Object {
```

```
    private String name;  
    private String ssn;
```

```
    public Employee(final String name, final String ssn) {  
        this.name = name;  
        this.ssn = ssn;  
    }
```

.....

```
@Override
```

```
    public String toString() {  
        return "(" + getName() + ", " + getSsn() + ")";  
    }
```

```
}
```

Se uma classe não derivar explicitamente de outra, derivará implicitamente da classe object, que está no topo da hierarquia de classes do Java.

Agora percebe-se! A classe object declara a operação toString() e define imediatamente um correspondente método. Esta é uma sua especialização.

Métodos finais

- Classe derivada não é obrigada a fornecer método para operação da classe base
- Classe base pode proibir às classes derivadas a sobreposição de um seu método, que se dirá ser um método **final**
- Razão para um método ser final:
 - Programador que forneceu o método na classe base entendeu que classes derivadas não deveriam poder especializar o modo de funcionamento desse método

Acesso à classe base

```
public class Base {  
    public String className() {  
        return "Base";  
    }  
}  
  
public class Derived extends Base {  
    @Override  
    public String className() {  
        return "Derived";  
    }  
  
    public void testCalls() {  
        Base base = (Base)this;  
        out.println("Through this: " + this.className());  
        out.println("Through base: " + base.className());  
        out.println("Through super: " + super.className());  
    }  
}
```

Through this: Derived
Through base: Derived
Through super: Base
—

Análise: conceitos

- Veículo
- Motociclo
- Automóvel
- Honda NX 650
- Audi TT

Vehicle

Motorcycle

Car

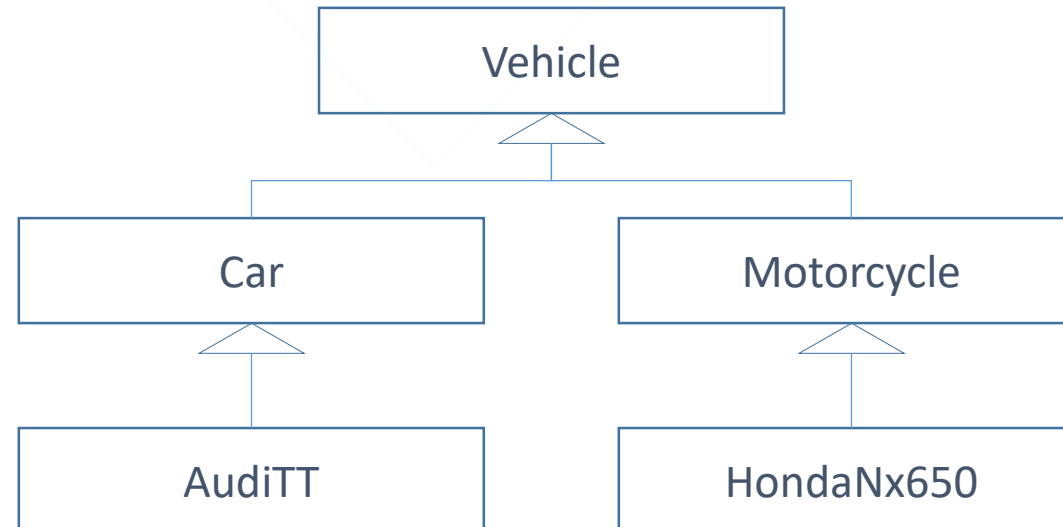
HondaNx650

AudiTT

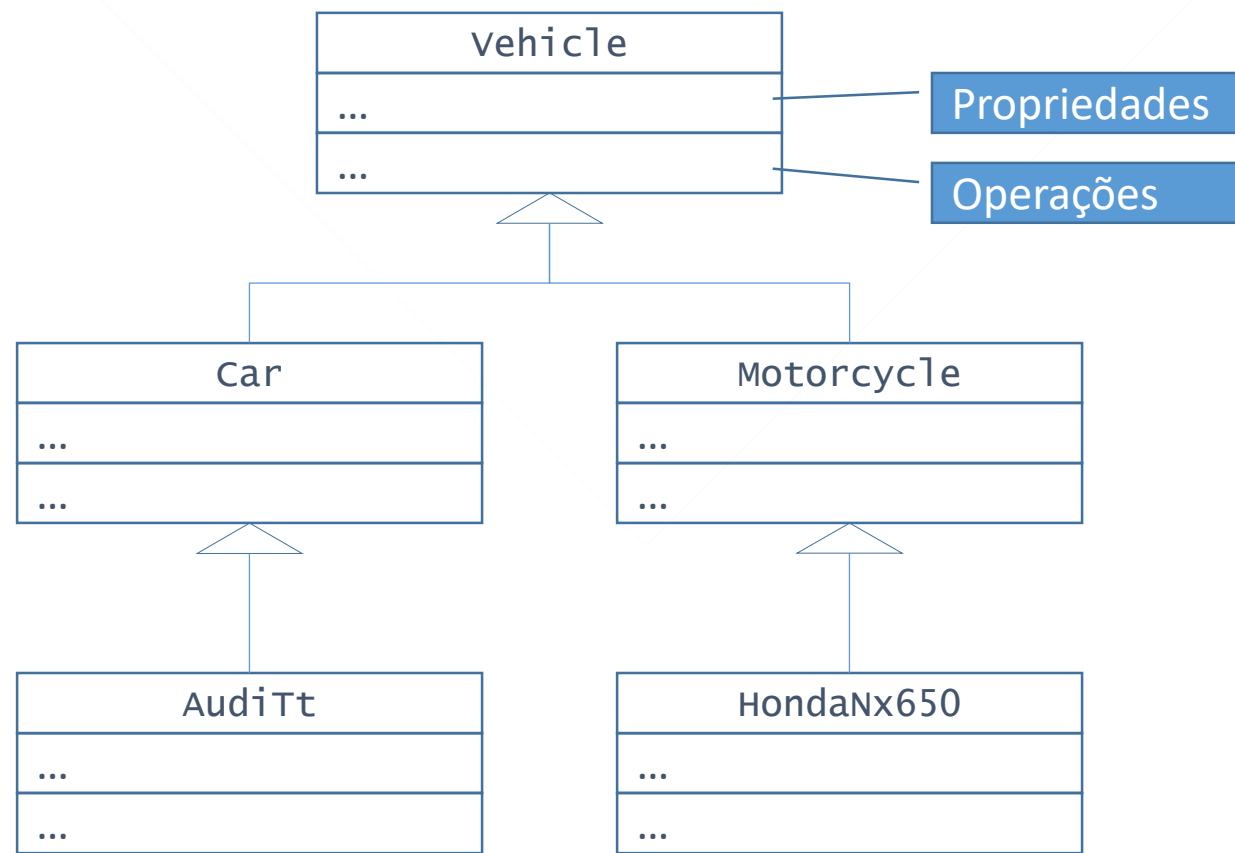
Análise: relações

- Um Automóvel é um Veículo
- Um Motociclo é um Veículo
- Uma Honda NX 650 é um Motociclo
- Um Audi TT é um Automóvel

Pode refinar-se o dicionário ou glossário do domínio, acrescentando as relações entre conceitos.

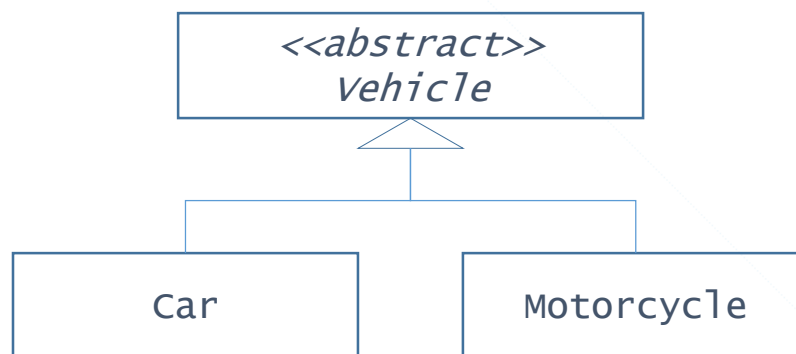


Desenho (Diagrama de Classes)



Análise e desenho

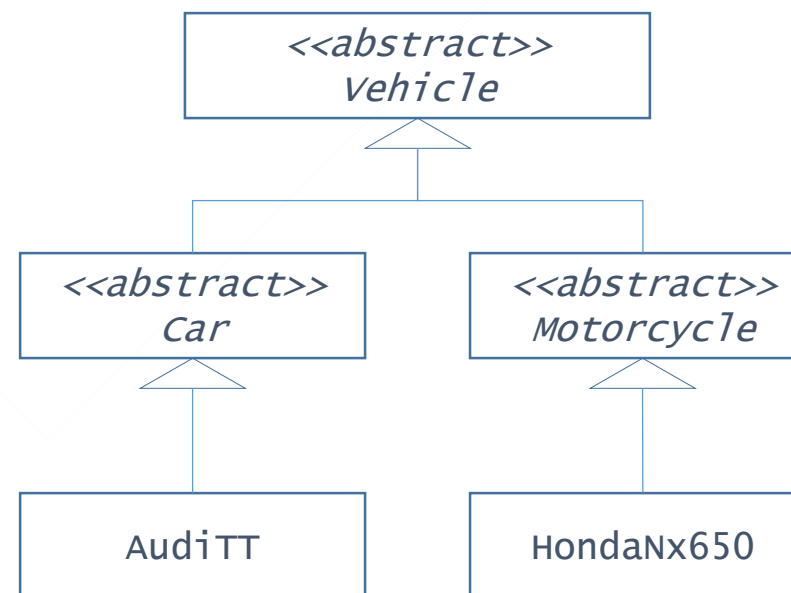
Hipótese 1



É boa prática que as classes concretas sejam folhas na hierarquia.

As classes abstractas, correspondentes aos conceitos abstractos, têm o nome em itálico.

Hipótese 2



Implementação: hipótese 1

```
public abstract class vehicle {  
    ...  
}  
  
public class Car extends vehicle {  
    ...  
}  
  
public class Motorcycle extends vehicle {  
    ...  
}
```

Implementação: hipótese 2

```
public class Vehicle {  
    ...  
}
```

```
public class Car extends Vehicle {  
    ...  
}
```

```
public class Motorcycle extends Vehicle {  
    ...  
}
```

```
public class HondaNx650 extends  
    Motorcycle {  
    ...  
}
```

```
public class AudiTT extends Car {  
    ...  
}
```

Classes abstratas

- Uma operação com qualificador `abstract` é uma simples **declaração** da operação
- Uma operação sem qualificador `abstract` inclui também a **implementação**
- Uma classe com uma operação abstrata tem de ser uma **classe abstracta**
- Uma classe é abstrata se tiver o qualificador `abstract`

Classes abstratas

- Uma classe não abstrata diz-se uma **classe concreta**
- Uma classe abstrata não pode ser instanciada, i.e., não se podem construir objectos de uma classe abstrata
- Uma classe derivada directamente de uma classe abstrata só poderá ser concreta se implementar cada uma das operações abstratas da classe abstrata

Análise: conceitos

- Figura
- Forma (abstrata)
- Círculo
- Quadrado

Figure

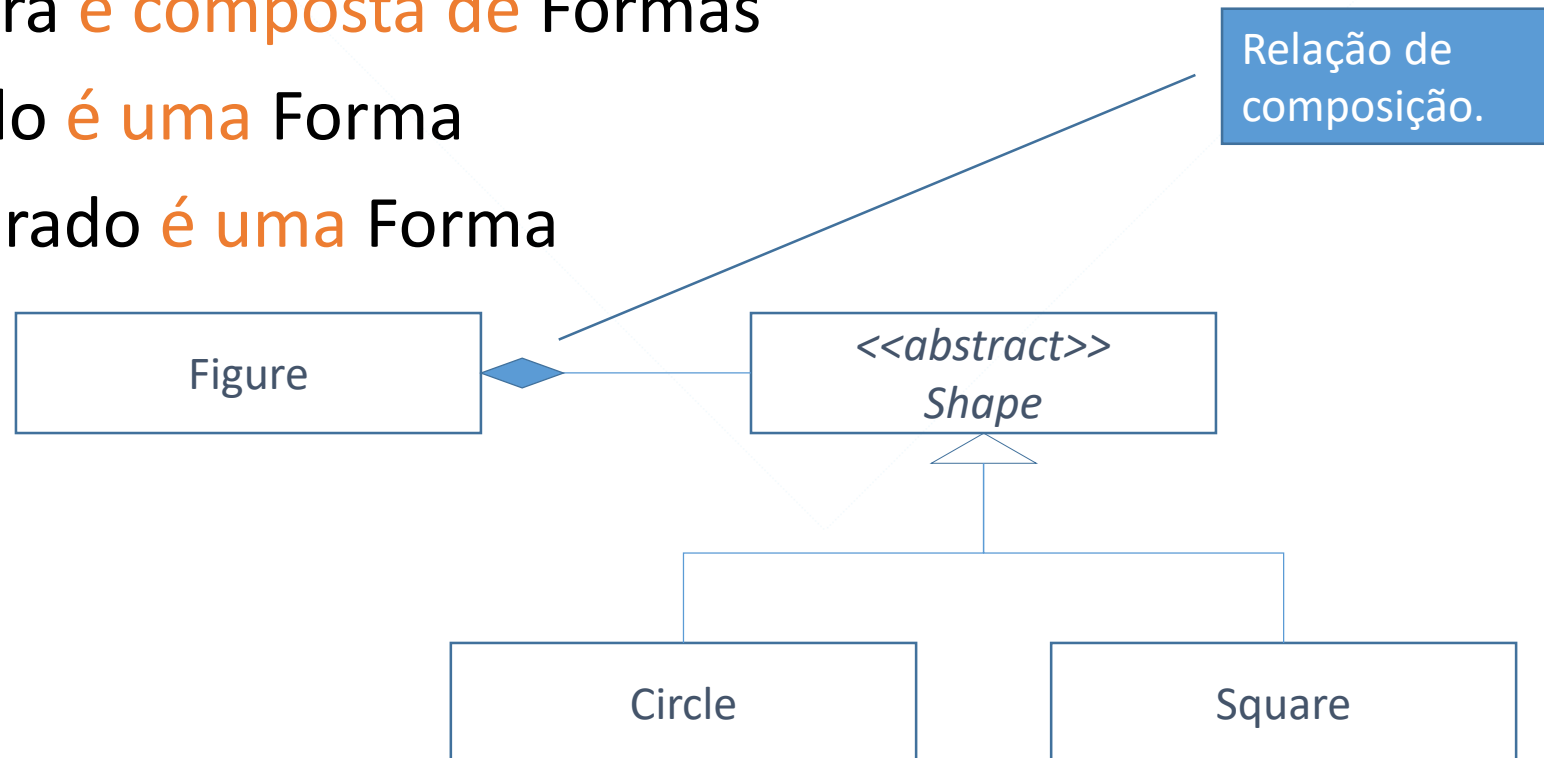
Shape

Circle

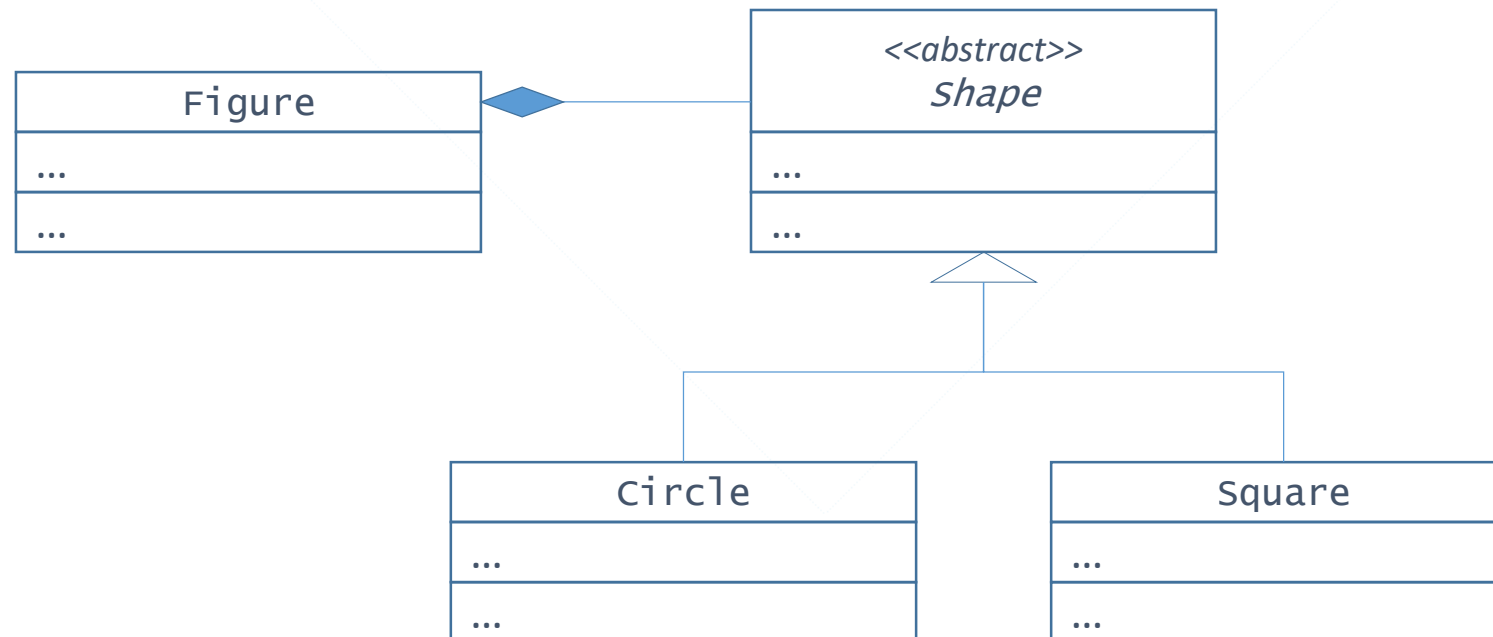
Square

Análise: relações

- Uma Figura é composta de Formas
- Um Círculo é uma Forma
- Um Quadrado é uma Forma



Desenho



Implementação

```
public class Figure {  
    private ArrayList<Shape> shapes;  
    ...  
}  
  
public abstract class Shape {  
    ...  
}  
  
public class Circle extends Shape {  
    ...  
}  
  
public class Square extends Shape {  
    ...  
}
```

Implementação: Shape

```
public abstract class Shape {  
    private Point position;  
  
    public Shape(final Point position) {  
        this.position = position;  
    }  
  
    public final Point getPosition() {  
        return position;  
    }  
  
    public abstract double getArea();  
    public abstract double getPerimeter();  
    public abstract Box getBoundingBox();  
  
    public void moveTo(final Point newPosition) {  
        position = newPosition;  
    }  
}
```

Qual a área de uma
“forma”??

Operações abstratas,
ou seja, operações
sem qualquer
implementação
disponível até este
nível da hierarquia.

Implementação: Circle

```
public class Circle extends Shape {  
  
    private double radius;  
  
    public Circle(final Point position,  
                  final double radius) {  
        super(position);  
        this.radius = radius;  
    }  
  
    public final double getRadius() {  
        return radius;  
    }  
}
```

Um circle é uma shape e a classe Circle herda a implementação da classe Shape.

É necessário apenas um atributo adicional, correspondente a uma das duas propriedades de um círculo (o raio), já que a posição do centro é herdada da classe Shape.

Uma ajudinha da classe base...

Implementação: Circle

```
@Override  
public double getArea() {  
    return Math.PI * getRadius() * getRadius();  
}
```

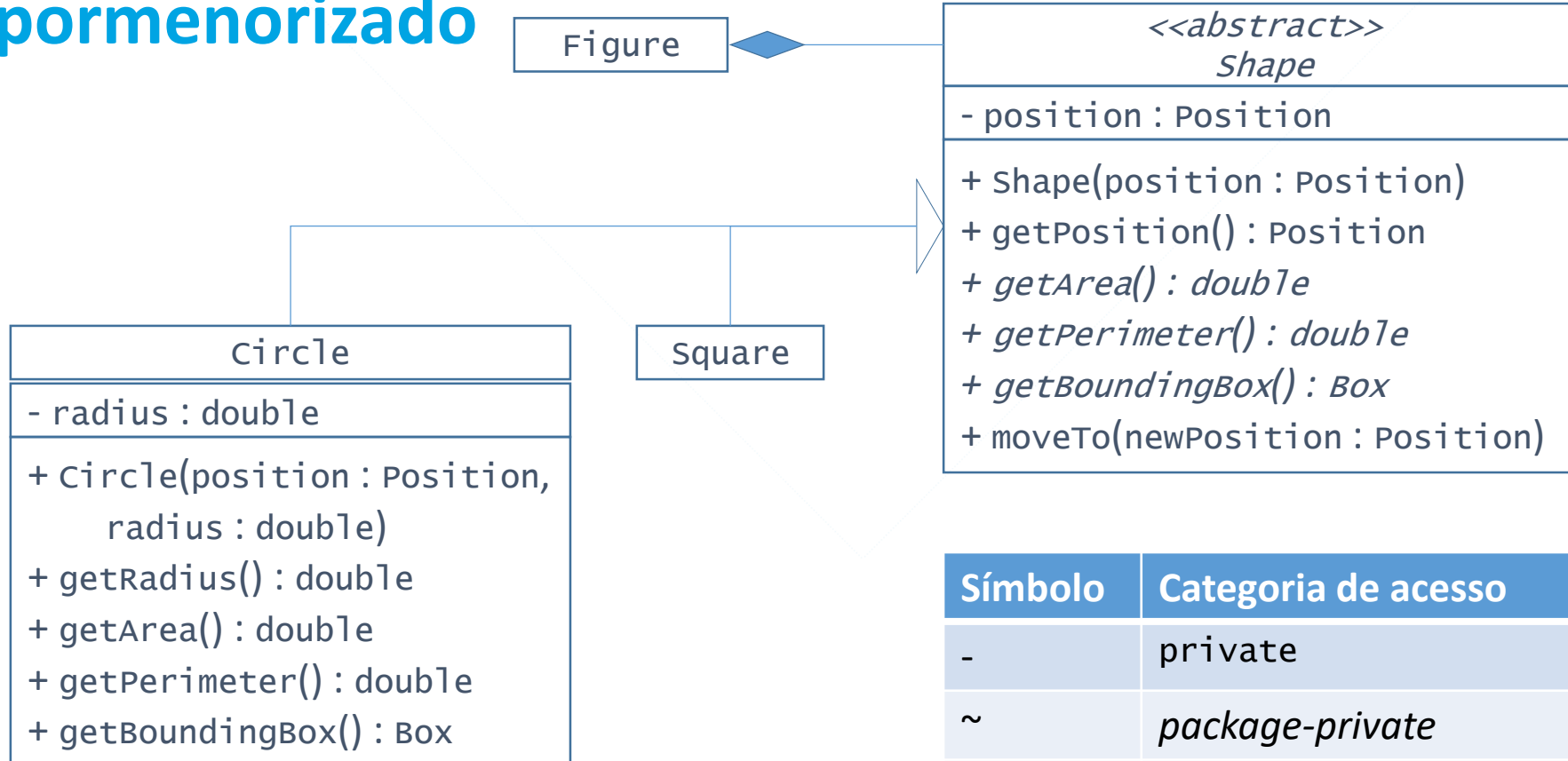
Qual a área de um círculo? Fácil, $\pi \times r^2$.

```
@Override  
public double getPerimeter() {  
    return 2.0 * Math.PI * getRadius();  
}
```

Fornece-se implementações, ou seja, métodos, para cada uma das operações abstractas da classe shape.

```
@Override  
public Box getBoundingBox() {  
    return new Rectangle(  
        new Point(getPosition().getX() - getRadius(), getPosition().getY() - getRadius()), new Size(2.0 * getRadius(), 2.0 * getRadius())  
    );  
}
```

Desenho pormenorizado



Símbolo	Categoria de acesso
-	private
~	package-private
#	protected
+	public

Desenho pormenorizado

